# 「Javaアプリケーション脆弱性事例調査資料」について

- この資料は、Javaプログラマである皆様に、脆弱性を身近な問題として感じてもらい、セキュアコーディングの重要性を認識していただくことを目指して作成しています。

- 「Javaセキュアコーディングスタンダード CERT/Oracle版」と合わせて、セキュアコーディングに関する理解を深めるためにご利用ください。

JPCERTコーディネーションセンター
セキュアコーディングプロジェクト
secure-coding@jpcert.or.jp

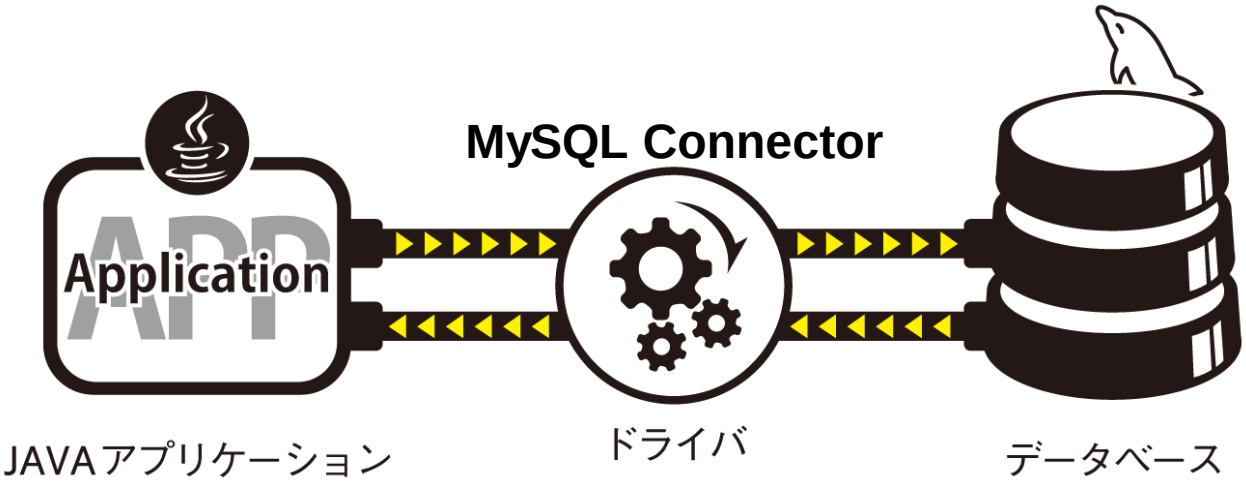
# MySQL Connector/J における SQL インジェクションの脆弱性

JVN#59748723

JVNDB-2009-000050

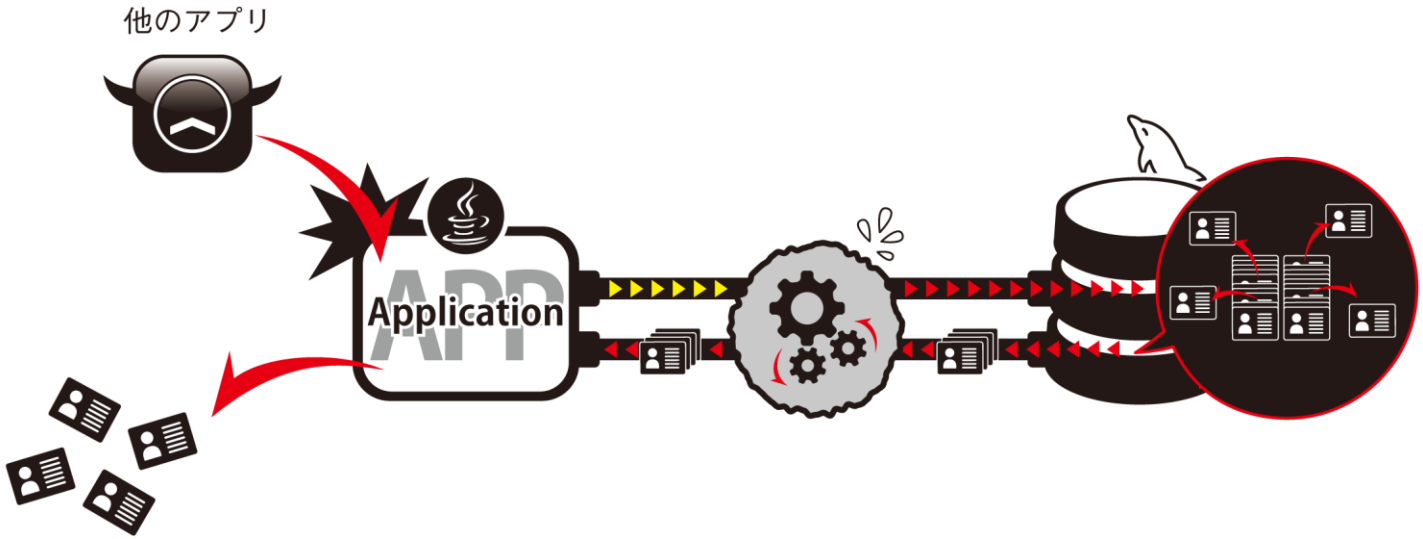
# MySQL Connectorとは

- MySQLデータベースにアクセスするための Java アプリケーション用ドライバソフトウェア
- MySQLを利用するJavaアプリケーションを簡単に作成できる

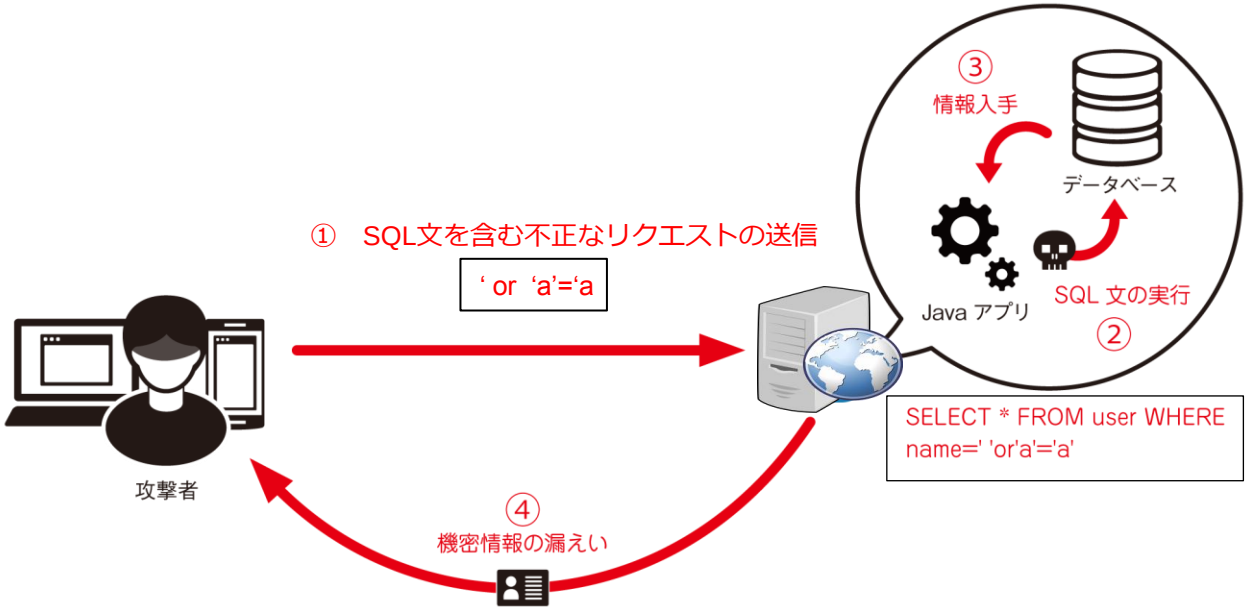


# 脆弱性の概要

- MySQL Connector/J にはSQLクエリ文字列の処理に不備があり、SQLインジェクションが可能となる脆弱性が存在する。
- Javaアプリケーションがプリペアードステートメント等の適切な処理を実装している場合でもSQLインジェクションが可能。



# SQLインジェクションとは

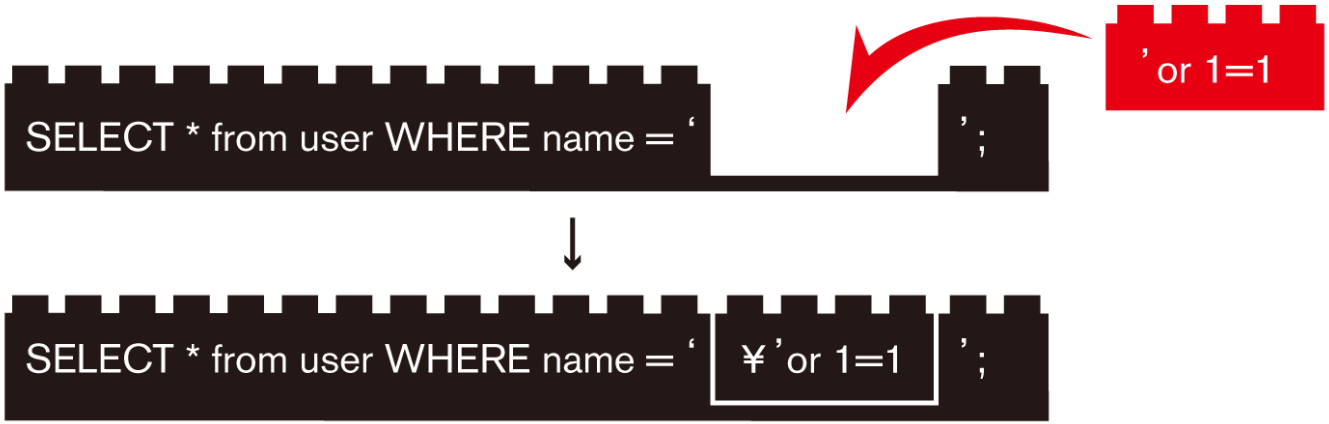



入力値を元にSQL文を動的に生成しているアプリケーションに対し、細工した入力を与えることで(アプリケーションの意図しないような) 任意のSQL文を挿入/実行すること。

データベース上のデータの**漏えい**や**改ざん**、**破壊**などの被害を受ける可能性がある。

# プリペアードステートメントとは

- 事前に定義したSQL文のプレースホルダ(予約場所)に入力データを割り当てる機能。
- プリペアドステートメントを利用すると、入力データは数値定数や文字列定数として組み込まれ、文字列として扱われることになる。
- 一般的にSQLインジェクション対策として使用される。



# プリペアードステートメントとは: サンプルコード

```
1: Connection conn = DriverManager.getConnection(
        "jdbc:mysql://192.168.xxx.xxx/?characterEncoding=Windows-31J",“id", "password");
2: String input = request.getParameter("name");
3: String sql = "SELECT * FROM user WHERE name=?";
4: PreparedStatement ps = conn.prepareStatement(sql);
5: ps.setString(1, input); //  1 は 1 番目のプレースホルダを表す
6: ResultSet resultSet = ps.executeQuery();
```

**解説**

- **1行目：データベースと接続する**
- **2行目：リクエスト中のパラメータnameの値を取得して変数inputに格納する**
- **3行目：「?」文字がプレースホルダ**
- **4行目：SQL文のプリコンパイルを行う。**
- **5行目：プレースホルダと変数の関連付けを行う。この例では、1番目のプレースホルダに変数inputを設定する。**
- **6行目：SQL文を実行し結果を得る。**

# MySQL Connector の処理内容

前述のサンプルコードを使用して処理を解説する。

```
1: Connection conn = DriverManager.getConnection(
     "jdbc:mysql://192.168.xxx.xxx/?characterEncoding=Windows-31J",“id", "password");
2: String input = request.getParameter("name");
3: String sql = "SELECT * FROM user WHERE name=?";
4: PreparedStatement ps = conn.prepareStatement(sql);
5: ps.setString(1, input);
6: ResultSet resultSet = ps.executeQuery();
```

**MySQL Connector側で処理が行われる箇所**
- **com.mysql.jdbc.Connectionクラス**
- **com.mysql.jdbc.PreparedStatementクラス**

# MySQL Connector の処理内容

```
1: Connection conn = DriverManager.getConnection(
     "jdbc:mysql://192.168.xxx.xxx/?characterEncoding=Windows-31J",“id", "password");
2: String input = request.getParameter("name");
3: String sql = "SELECT * FROM user WHERE name=?";
4: PreparedStatement ps = conn.prepareStatement(sql);
5: ps.setString(1, input);
6: ResultSet resultSet = ps.executeQuery();
```

Point!!

サンプルコードが実行された場合のMy SQL Connecter側の処理フロー

① 1行目でデータベースに接続する。

② **5行目のPreparedStatement::setStringメソッド内で第2引数inputの文字列のエスケープが実行される。**

③ **同じくPreparedStatement:: setStringメソッド内で文字列がByte列に変換され、プレースホルダに値が設定される。**

④ 6行目でSQLクエリが実行される。

# MySQL Connector の処理
# ①データベースへの接続

```
1: Connection conn = DriverManager.getConnection(
     "jdbc:mysql://192.168.xxx.xxx/?characterEncoding=Windows-31J",“id", "password");
2: String input = request.getParameter("name");
3: String sql = "SELECT * FROM user WHERE name=?";
4: PreparedStatement ps = conn.prepareStatement(sql);
5: ps.setString(1, input);
6: ResultSet resultSet = ps.executeQuery();
```

データベースに接続する。パラメータcharacterEncoding で
アプリケーションで使用する文字エンコーディングを指定する。

# MySQL Connector の処理
## ②PreparedStatement::setStringで第2引数のエスケープ

```
1: Connection conn = DriverManager.getConnection(
     "jdbc:mysql://192.168.xxx.xxx/?characterEncoding=Windows-31J",“id", "password");
2: String input = request.getParameter("name");
3: String sql = "SELECT * FROM user WHERE name=?";
4: PreparedStatement ps = conn.prepareStatement(sql);
5: ps.setString(1, input);
6: ResultSet resultSet = ps.executeQuery();
```

**変数inputをリクエストから受け取り、PreparedStatement::setString メソッドの第2引数として渡す。**

# MySQL Connector の処理
## ②PreparedStatement::setStringで第2引数のエスケープ

```
public class PreparedStatement extends ・・・・・・・{
    public void setString(int parameterIndex, String x) throws SQLException {
        :
        StringBuffer buf = new StringBuffer((int) (x.length() * 1.1));

        for (int i = 0; i < stringLength; ++i) {
            char c = x.charAt(i);
            switch (c) {
            case 0: /* Must be escaped for 'mysql' */
                buf.append('¥¥');
                buf.append('0');
                break;
            :
            case '¥¥':
            :
            case '¥'':
            :
        default:
            buf.append(c);
        }
```

setStringメソッドの第2引数のxがプレースホルダにセットされる文字列。

Xから1文字ずつ取り出して、特殊文字に対してエスケープ処理を行う。

# MySQL Connector の処理
## ②PreparedStatement::setStringで第2引数のエスケープ

- エスケープ処理のまとめ

| 入力文字列 | エスケープ処理後の文字列 |
|---|---|
| NULL | ¥0 |
| ¥n | ¥n |
| ¥r | ¥r |
| ¥ (スラッシュ) | ¥¥ |
| ‘ (シングルクオート) | ¥’ |
| “ (ダブルクオート) | ¥” |
| ¥032 | ¥Z |
| 上記以外 | 特に処理なし |

setStringメソッドに渡される値が「’ or 1=1」の場合は
上記のエスケープ処理によって「 ¥’ or 1=1 」となる。

# MySQL Connector の処理
## ③PreparedStatement::setStringでByte列への変換

```
public class PreparedStatement extends com.mysql.jdbc.StatementImpl
    implements java.sql.PreparedStatement {

    public void setString(int parameterIndex, String x) throws SQLException {
        :
        parameterAsString = buf.toString();
        byte[] parameterAsBytes = null;

        parameterAsBytes = StringUtils.getBytes(parameterAsString,
                                this.charConverter, this.charEncoding, this.connection
                                .getServerCharacterEncoding(), this.connection
                                .parserKnowsUnicode());
        :

        setInternal(parameterIndex, parameterAsBytes);
```

エスケープ処理後の文字列がString型変数parameterAsStringに保存される。

Windows-31J

StringUtils::parameterAsStringメソッドで文字列をByte列に変換する。その際に指定されるcharEncodingは接続時に指定した文字コード(Windows-31J)が設定される。

# MySQL Connector の処理
## ③PreparedStatement::setStringでByte列への変換

■ StringUtils.getBytesメソッドによるByte列への変換

StringUtils.getBytesメソッドでは内部的にString.getBytesメソッドが呼び出されている。

**StringUtils.java**

setStringメソッドの第2引数

Windows-31J

```
public static final byte[] getBytes(String s,
                    SingleByteCharsetConverter converter, String encoding,
                    String serverEncoding, boolean parserKnowsUnicode)
                    throws SQLException {
    :
    b = s.getBytes(encoding);
    :
    return b;
```

■ 変換対象の文字列（第1引数s）が「 ¥’ or 1=1 」の場合

⇒ **[92,39, 32, 111,114, 32, 49,61,49]** というByte列に変換される。

¥ ’ space o r space 1 = 1

# MySQL Connector の処理
## ③PreparedStatement::setStringでByte列への変換

```
public class PreparedStatement extends com.mysql.jdbc.StatementImpl
    implements java.sql.PreparedStatement {

        public void setString(int parameterIndex, String x) throws SQLException {
                :
            parameterAsString = buf.toString();
            byte[] parameterAsBytes = null;

            parameterAsBytes = StringUtils.getBytes(parameterAsString,
                                this.charConverter, this.charEncoding, this.connection
                                .getServerCharacterEncoding(), this.connection
                                .parserKnowsUnicode());
                :

            setInternal(parameterIndex, parameterAsBytes);
```

**com.mysql.jdbc.PreparedStatement.setInternal メソッドで、先ほど変換したByte列をプリペアードステートメントにセットする。**

# MySQL Connector の処理
## ④SQLクエリの実行

SQLが実行される。結果として実行されるSQLは・・・

■ **変数inputが「test」の場合**
実行されるSQLは
**「 SELECT * FROM user WHERE name='test'」**
となる。

■ **変数inputが「' or 1=1」の場合**
実行されるSQLは
**「 SELECT * FROM user WHERE name='¥' or 1=1'」**
となり、変数inputに含まれるシングルクオートがエスケープされる。

# MySQL Connector の処理のまとめ

①データベースへの接続

②PreparedStatement::setStringメソッド内で引数の文字列のエスケープ

「**’ or 1=1**」　→　「**¥’ or 1=1**」

③PreparedStatement:: setStringメソッド内で文字列をByte列に変換

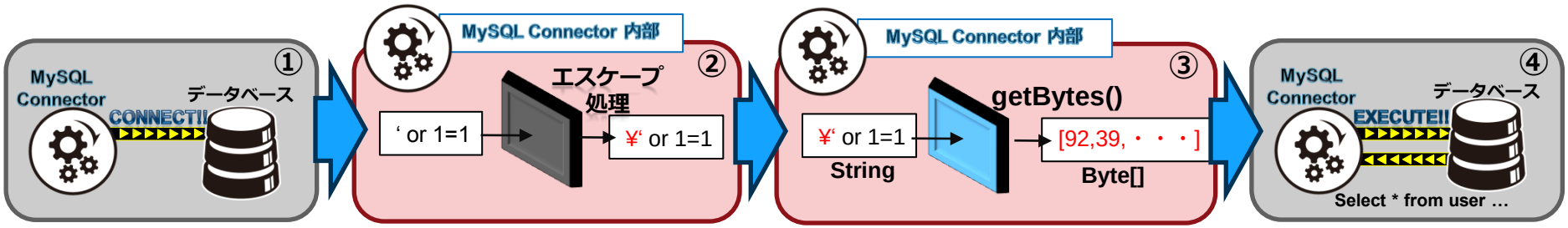
「 **¥’ or 1=1** 」 → [92,39,32,111,114,32,49,61,49]
¥ ’ space o r space 1 = 1

プレースホルダへのセット

[92,39,32,111,114, 32, 49,61,49]
¥ ’ space o r space 1 = 1

SELECT * FROM user WHERE name=‘ **?** ‘

④SQLの実行 **SELECT * FROM user WHERE name=‘¥’ or 1=1’**



# 攻撃コード

```
1: Connection conn = DriverManager.getConnection(
   "jdbc:mysql://192.168.xxx.xxx/?characterEncoding=Windows-31J",“id", "password");
2: String sql = "SELECT * FROM user WHERE name=?";
3: PreparedStatement ps = conn.prepareStatement(sql);
4: ps.setString(1, "¥u00a5' or 1 = 1#");
5: ResultSet resultSet = ps.executeQuery();
```

■攻撃コードのポイント

- 4行目のsetStringメソッドの引数(プレースホルダに入れる値)にUNICODEで‘¥’ (YEN SIGN) に該当する**U+00A5**を挿入し、その後に「**’ or 1=1#**」という値を渡している。
- 1行目のデータベース接続時の文字エンコーディングとして「**Windows-31J**」を指定しており、異なる文字エンコーディングを使用した文字列が含まれていることになる。

# 攻撃コード

サンプルコードが実行された場合のMy SQL Connecter側の処理フロー

① 1行目でデータベースに接続する。

② **5行目のPreparedStatement::setStringメソッド内で第2引数inputの文字列のエスケープが実行される。**

③ **同じくPreparedStatement::setStringメソッド内でgetBytesメソッドにより文字列がByte列に変換され、プレースホルダに値が設定される。**

④ 6行目でSQLクエリが実行される。

**②と③で脆弱性の原因となる処理が実行される**

# 攻撃コードが実行された際の処理
# ②PreparedStatement::setStringで第2引数のエスケープ

¥' or 1 = 1#
※¥はUNICODEの'¥'である¥u00a5

```
public class PreparedStatement extends ・・・・・・・・{
    public void setString(int parameterIndex, String x) throws SQLException {
        :
        StringBuffer buf = new StringBuffer((int) (x.length() * 1
        for (int i = 0; i < stringLength; ++i) {
            char c = x.charAt(i);
            switch (c) {
            case 0: /* Must be escaped for 'mysql' */
                buf.append('¥¥');
                buf.append('0');
                break;
            :
            case '¥¥':
            :
            default:
                buf.append(c);
        }
```

setStringメソッドの第2引数のxがプレースホルダにセットされる。

Xから1文字ずつ取り出して、特殊文字に対してエスケープ処理を行う。

しかし、 UNICODEの'¥'である¥u00a5はエスケープされない。
そのため、エスケープ処理の結果は
「¥' or 1 = 1#」 → 「¥¥' or 1 = 1#」
となる！！

※¥はUNICODEの'¥'である¥u00a5

# 攻撃コードが実行された際の処理
## ③PreparedStatement::setStringでgetBytesメソッドによるByte列への変換

```
public class PreparedStatement extends com.mysql.jdbc.StatementImpl
implements
java.sql.PreparedStatement {

    public void setString(int parameterIndex, String x) throws SQLException {
        :
        parameterAsString = buf.toString();
        byte[] parameterAsBytes = null;

        parameterAsBytes = StringUtils.getBytes(parameterAsString,
                this.charConverter, this.charEncoding, this.connection
                .getServerCharacterEncoding(), this.connection
                .parserKnowsUnicode());
        :

        setInternal(parameterIndex, parameterAsBytes);
```

¥¥' or 1 = 1#
※¥はUNICODEの'¥'である¥u00a5

エスケープ処理後の文字列がString型変数parameterAsStringに保存される。

Windows-31J

StringUtils::parameterAsStringメソッドで文字列をByte列に変換する。
その際に指定されるcharEncodingは接続時に指定したエンコーディング(Windows-31J)が設定される。

# 攻撃コードが実行された際の処理
## ③PreparedStatement::setStringでgetBytesメソッドによるByte列への変換

■ StringUtils.getBytesメソッドによるByte列への変換
StringUtils.getBytesの内部ではString.getBytesメソッドを使っている。

**StringUtils.java**

¥¥' or 1 = 1#
※¥はUNICODEの'¥'である¥u00a5

Windows-31J

```
public static final byte[] getBytes(String s,
                    SingleByteCharsetConverter converter, String encoding,
                    String serverEncoding, boolean parserKnowsUnicode)
                    throws SQLException {
    :
    b = s.getBytes(encoding);
    :
    return b;
```

■ 攻撃コードが実行された場合
**getBytesメソッドでU+00A5はShift JISの'¥'として変換されてしまう!!**
「¥¥' or 1 = 1#」 → [92,92,39, 32,111,114, 32,49,61,49,35]
¥ ¥ ' space o r space 1 = 1 #
※¥はUNICODEの'¥'である¥u00a5

本来のバイト列は[165,92,39,32,111,114,32,49,61,49,35]

# 攻撃コードが実行された際の処理
## ③PreparedStatement::setStringでgetBytesメソッドによるByte列への変換

```
public class PreparedStatement extends com.mysql.jdbc.StatementImpl
    implements  java.sql.PreparedStatement {

        public void setString(int parameterIndex, String x) throws SQLException {
                :
            parameterAsString = buf.toString();
            byte[] parameterAsBytes = null;

            parameterAsBytes = StringUtils.getBytes(parameterAsString,
                                this.charConverter, this.charEncoding, this.connection
                                .getServerCharacterEncoding(), this.connection
                                .parserKnowsUnicode());
                :

            setInternal(parameterIndex, parameterAsBytes);
```

[92,92,39, 32,111,114,32,49,61,49,35]
¥ ¥ ’ space o r space 1 = 1 #

com.mysql.jdbc.PreparedStatement.setInternalメソッドで、
先ほど変換したByte列をプリペアードステートメントにセットする。

# 攻撃コードが実行された際の処理

■プレースホルダへのセット
変換されたByte列がプレースホルダにセットされる。

[92,92,39,32,111,114,32,49,61,49,35]
¥ ¥ ’ space o r space 1 = 1 #

SELECT * FROM user WHERE name=‘ ? ‘

**SELECT * FROM user WHERE name=‘¥¥’ or 1=1#’**

**SQL文のWhere句は、**
**「nameが¥(バックスラッシュ)と等しい、あるいは、1が1と等しい」という条件になり、本来実行されるべきクエリから内容が変更されてしまった!!　(#はMySQLでのコメント文字であり、これ以降は無視される.)**

# 攻撃コードが実行された際の処理

①データベースへの接続

②PreparedStatement::setStringメソッド内で引数の文字列のエスケープ

「¥’ or 1=1」　→　「¥¥’ or 1=1」　※¥はUNICODEの’¥’である¥u00a5

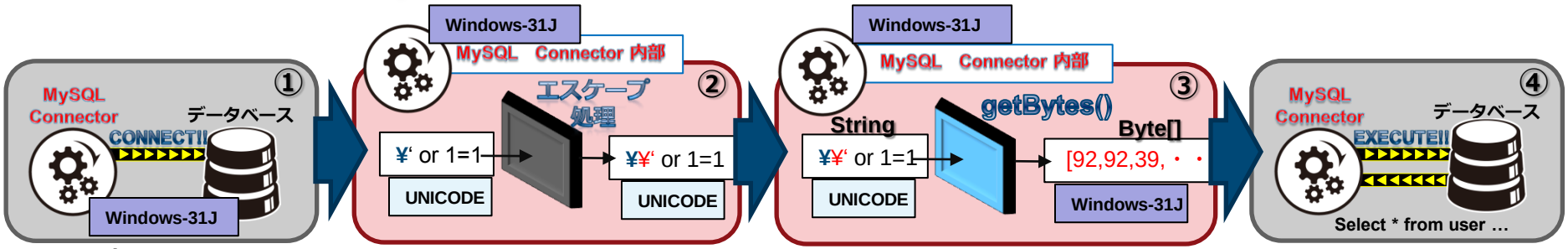
③PreparedStatement:: setStringメソッド内で文字列→Byte列に変換

「 ¥¥’ or 1=1 」　→　[92,92,39, 32,111,114,32,49,61,49]
¥ ¥ ’ space o r space 1 = 1

プレースホルダへのセット

SELECT * FROM user WHERE name=‘ ? ‘

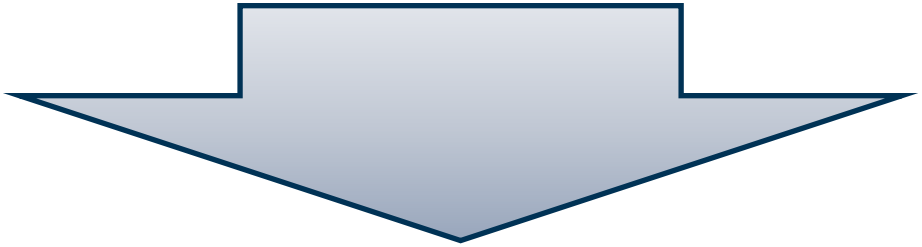
④SQLの実行　**SELECT * FROM user WHERE name=‘¥¥’ or 1=1#’**



# 問題点

- 今回のアプリケーションにおける具体的な問題点

  指定されたエンコーディングへの変換の**前に**エスケープ処理を行っており、適切なエスケープ処理ができていなかった。



以下のコーディングガイドに違反している!!
**「IDS01-J. 文字列は検査するまえに標準化する」**
**「IDS13-J. ファイル入出力やネットワーク入出力の両端で互換性のある文字エンコーディングを使う」**

**⇒その結果、エスケープ処理をバイパスしてSQLインジェクションが成立する形になっていた。**

# 問題点

- 問題点に対してどうすべきだったか。

  **文字エンコーディングの変換をした後で文字列のエスケープ処理を実施すべきであった。**

# 修正版コード

この脆弱性対応はバージョン5.1.8で行われている。

サンプルコードが実行された場合のMy SQL Connecter側の処理フロー

① 1行目でデータベースに接続する。

② **5行目のPreparedStatement::setStringメソッド内で第2引数inputの文字列のエスケープが実行される。**

③ **同じくPreparedStatement:: setStringメソッド内でgetBytesメソッドにより文字列がByte列に変換され、プレースホルダに値が設定される。**

④ 6行目でSQLクエリが実行される。

**②の処理に修正が加えられている。**

# 修正版コード

## ②PreparedStatement::setStringメソッド内で第2引数のエスケープ

```
public class PreparedStatement extends com.mysql.jdbc.StatementImpl
   implements  java.sql.PreparedStatement {
      public void setString(int parameterIndex, String x) throws SQLException {

            switch (c) {
                  :
               case '¥u00a5':
               case '¥u20a9':
               // escape characters interpreted as backslash by mysql
                        if(charsetEncoder != null) {
                                 CharBuffer cbuf = CharBuffer.allocate(1);
                                 ByteBuffer bbuf = ByteBuffer.allocate(1);
                                 cbuf.put(c);
                                 cbuf.position(0);
                                 charsetEncoder.encode(cbuf, bbuf, true);
                                 if(bbuf.get(0) == '¥¥') {
                                          buf.append('¥¥');
                                     :
                                  buf.append(c);
```

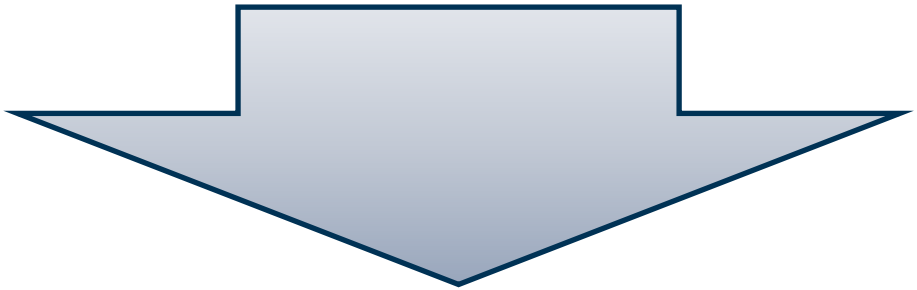
**setStringメソッドにて¥u00a5に対するエスケープ処理が追加されている。**

# 修正の内容

- U+00A5に対するエスケープ処理が追加されている。
  ⇒指定された文字エンコーディングに変換し、変換結果が￥(バックスラッシュ)だった場合はもうひとつ￥(バックスラッシュ)を追加してエスケープする。
- U+20a9(ウォンマーク）についても同様の脆弱性が発生するため、エスケープ処理が追加されている。

# 修正版コードに対する考察

- 問題となる文字にだけ処理を行う「ブラックリスト」的なアプローチ。同様の問題が発生するケースが他に存在するかも



- U+00A5やU+20a9だけでなく、文字列全体に同様の処理を行うべきでは?
- case句の前に入力文字列全体を指定された文字エンコーディングに変換する。(次ページにサンプルを記載)

# もう一つの修正案
## エスケープ処理の前に文字エンコーディング変換

```
public class PreparedStatement extends ・・・・・・・{
    public void setString(int parameterIndex, String x) throws SQLException {
        :
        StringBuffer buf = new StringBuffer((int) (x.length() * 1.1));
        byte[] bytex = x.getBytes(common_encoding);
        for (int i = 0; i < stringLength; ++i) {
            byte c = bytex[i];
            switch (c) {
            case 0: /* Must be escaped for 'mysql' */
            :
            case 92:
            :
            case 39:
            :
            default:
                buf.append(c);
            }
```

エスケープ処理の前に文字エンコーディングを指定してバイト列に変換する。変数common_encodingはアプリ内で使用する文字エンコーディングを指定。

その後 byte型で比較を行い、エスケープ処理を行う。これにより、文字エンコーディングの不整合を悪用したエスケープ処理のバイパスはできなくなる。

# 文字エンコーディング不整合による処理の不備

文字エンコーディングに関連する処理の不備で発生した脆弱性は他にも見つかっている。

JVN ipediaで「文字コード」で検索をかけると複数の脆弱性がヒットする。(2012年11月時点で24件)

**JVN ipedia の検索結果より抜粋**

| JVN ID | タイトル |
|---|---|
| JVNDB-2006-000306 | PostgreSQL における特定のマルチバイト文字コードによる SQL インジェクションの脆弱性 |
| JVNDB-2012-001321 | 複数の Siemens 製品の HMI Web サーバにおける任意のメモリロケーションからデータを読まれる脆弱性 |
| JVNDB-2007-000398 | SquirrelMail におけるクロスサイトスクリプティングの脆弱性 |

# まとめ

■ この脆弱性から学べるプログラミングの注意点
- •文字列に対して複数の処理を行う場合、処理の順序によっては目的とする効果が得られない場合がある
  - •今回のケースでは、SQLのメタ文字に対するエスケープ処理と文字エンコーディング変換処理

■ 上記への対策
- • アプリ内部の文字列処理では、まず最初に文字エンコーディングを統一するための前処理を入れる
- • 可能であれば、文字列に異なる文字エンコーディングのデータが含まれていた場合はエラーとする

## 著作権・引用や二次利用について

■本資料の著作権はJPCERT/CCに帰属します。

■本資料あるいはその一部を引用・転載・再配布する際は、引用元名、資料名および URL の明示をお願いします。

記載例

引用元：一般社団法人JPCERTコーディネーションセンター
Java アプリケーション脆弱性事例解説資料
MySQL Connector/J における SQL インジェクションの脆弱性
https://www.jpcert.or.jp/securecoding/2012/No.04_MySQL_Connector.pdf

■本資料を引用・転載・再配布をする際は、引用先文書、時期、内容等の情報を、JPCERT コーディネーションセンター広報(office@jpcert.or.jp)までメールにてお知らせください。なお、この連絡により取得した個人情報は、別途定めるJPCERT コーディネーションセンターの「プライバシーポリシー」に則って取り扱います。


本資料の利用方法等に関するお問い合わせ
JPCERTコーディネーションセンター
広報担当
E-mail：office@jpcert.or.jp

本資料の技術的な内容に関するお問い合わせ
JPCERTコーディネーションセンター
セキュアコーディング担当
E-mail：secure-coding@jpcert.or.jp